CONSTRAINED EXPRESSIONS: ADDING ANALYSIS CAPABILITIES TO DESIGN METHODS FOR CONCURRENT SOFTWARE SYSTEMS

George S. Avrunin
Laura K. Dillon
Jack C. Wileden
William E. Riddle

COINS Technical Report 85–13
May 1985
(Revised July 1985)

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

A version of this report is to appear in a special issue on Software Design Methods of the IEEE Transactions on Software Engineering.

G. S. Avrunin is with the Department of Mathematics and Statistics, University of Massachusetts, Amherst MA 01003.

L. K. Dillon was with the Software Development Laboratory. She is now with the Computer Science Department, University of California, Santa Barbara CA 93106.

W. E. Riddle is with software design & analysis, inc., Boulder CO 80303.

Abstract

We describe an approach to the design of concurrent software systems based on the *constrained expression* formalism. This formalism provides a rigorous conceptual model for the semantics of concurrent computations, thereby supporting analysis of important system properties as part of the design process. At the same time, our approach allows designers to use standard specification and design languages, rather than forcing them to deal with the formal model explicitly or directly. As a result, our approach attains the benefits of formal rigor without the associated pain of unnatural concepts or notations for its users.

The conceptual model of concurrency underlying the constrained expression formalism treats the collection of possible behaviors of a concurrent system as a set of sequences of events. The constrained expression formalism provides a useful closed-form description of these sequences. We have developed algorithms for translating designs expressed in a wide variety of notations into these constrained expression descriptions. We have also developed a number of powerful analysis techniques that can be applied to these descriptions.

In this paper, we describe the constrained expression formalism and these analysis techniques. We then describe the way this approach would be used in design, giving an example illustrating its use in conjunction with an Ada-like design language, and discuss present and future prospects for its automation and use.

Index terms: constrained expressions, design method, event-based, concurrent software systems, Ada-based design notation, analysis techniques

1. Introduction

Designing any concurrent software system, particularly a distributed system, is a complex and error-prone task. The main source of difficulty is the large number of subtle and often unexpected interactions that can occur among the various parts of an asynchronous system. The number and complexity of these interactions make it extremely hard to understand and accurately describe the properties of the system. To overcome these problems, designers need both suitably precise notations for describing system designs and their properties and also methods for rigorously analyzing the behavior of the system represented by a design.

A wide variety of techniques for describing concurrent systems have been proposed, ranging from specification and design languages to programming languages and graphical formalisms. These proposals differ in focus, in the types of objects they use to represent aspects of a concurrent system, and in the assumptions they make about general features of concurrent computation, such as the nature of communication between various parts of the system. Such fundamental issues as what types of interprocess communication and synchronization primitives should be provided in a development notation have yet to be resolved. The experience with programming languages suggests that, in fact, different notations, based on different primitives, may be best suited for use with different types of concurrent systems or for use at different stages of the development process. All of this indicates that no single design notation for concurrent systems is likely to receive widespread acceptance in the near future.

Even an ideal design notation, however, would not by itself solve the designer's most pressing problem, which is to understand the interactions among the separate parts of the concurrent system under development. The overwhelming complexity of large concurrent systems means that a design method must provide a means for rigorously reasoning about the behavior of the system, not just a suitable design notation. This seems to imply that a design method should be based on some sufficiently powerful and flexible mathematical formalism. At the same time, if it is to be of any practical value, it must be amenable to use by software developers who have little or no training in advanced mathematics or theoretical computer science.

In this paper, we describe an approach, based on our *constrained expression* formalism, that addresses these problems. With this approach, a developer can formulate a design using any of a wide variety of standard design notations. This design is then algorithmically translated into a constrained expression description, which provides an appropriate formal structure on which to base analysis. We have developed a number of powerful analysis techniques that can be applied to these descriptions. Developers using this approach can design systems and phrase questions about their behavior in terms of a congenial design notation and then answer those questions using analysis techniques based on the constrained expression representation. Thus the constrained expression approach can add analysis capabilities to design methods based on a variety of different notations.

The conceptual model of concurrency underlying the constrained expression formalism treats the collection of possible behaviors of a concurrent system as a set of sequences of events. In the next section we describe this event-based model and briefly discuss some other approaches using similar models. In section 3, we outline the constrained expression formalism. We describe our approach and give an example illustrating its use in section 4, and discuss our experience with the approach in section 5. Our assessment of the approach and our plans for its further development are presented in section 6. Finally, an appendix provides a more complete and rigorous presentation of some of the analysis outlined in section 4.

2. The Event-Based Perspective

The fundamental idea underlying our conceptual model of concurrent computation is that system behaviors can be viewed as sequences of events. These events can be of arbitrary complexity, depending on the system characteristics of interest and the level of system description under consideration. When analysing communication properties of a system, for example, one might view the system's behavior as consisting of "send message" and "receive message" events, while for studying other aspects of the system "communicate" and "compute" events might provide a more appropriate viewpoint. By associating an *event symbol* to each event, we can regard each possible behavior of the system as a string over the alphabet of event symbols. Sets of such strings, and properties of those sets, then become the primary objects of interest in assessing the possible behaviors of a concurrent system.

We use interleaving to represent concurrency. Thus, a string representing a possible behavior of a system that consists of a number of concurrently executing components is obtained by interleaving, or shuffling, strings representing the behaviors of the components. The events themselves are assumed to be indivisible and atomic, with only one event taking place at any particular time. Events which are to be explicitly regarded as overlapping in time can easily be represented by treating their initiation and termination as distinct atomic events.

Viewing a behavior of the system as a sequence of (non-overlapping) events obviously implies that the events comprising that behavior are totally ordered in time. It has been argued [6,12] that concurrency can best be represented in terms of a partial order on events, with those events not comparable in the order being regarded as potentially concurrent. This approach is, in fact, entirely compatible with the interleaved model of concurrency. If the relative order of occurrence of two events is not determined by the nature of the system, then, for each of the possible orders, there will be system behaviors in which the events occur in that order. By considering the set of all possible system behaviors, the appropriate partial order can be recovered, even though events in any single behavior occur serially.

The developer of a concurrent system is primarily concerned with two sets of event sequences: the set representing the possible behaviors of the system as it is currently designed, and the set representing the desired behaviors of the system. Analysis of the design is intended to determine if the two sets are identical, or at least if one is contained in the other. A first problem, then, is to produce from the design a precise description of the set of event sequences representing possible behaviors of the system. In the constrained expression formalism this is done by deriving a regular expression whose language is a larger set of event sequences and then "filtering" this set to remove the sequences that do not represent possible behaviors. This leads to a closed-form description of the set of possible behaviors of the system.

To see why such an indirect approach is convenient, consider a number of sequential processes making up a concurrent system. If the processes do not interact, the possible behaviors of the complete system are represented by the set of all possible interleavings of sequences representing behaviors of the component processes. But if the processes do interact, not all of these interleavings represent real behaviors. For example, suppose one process produces information that is used by a second. The event sequences representing behaviors of the first process will include symbols representing the production of this information and its transmission to the second process. The sequences representing behaviors of the second process will include symbols representing the receipt of the information and its subsequent use. When such sequences are interleaved, however, no relationship is maintained between events from the different sequences. Thus a symbol representing the receipt of a piece of information could occur in the interleaved string before the symbol representing the production of the information. The result of such an interleaving clearly does not represent a possible behavior of the system, and should be discarded.

It may seem that it would be easier to describe the event sequences representing possible behaviors of the system directly, rather than first describing the event sequences for each component process and then removing those interleavings that represent "illegal" or "impossible" behaviors. Our experience indicates, however, that producing such a direct description of the behaviors of a concurrent system is extremely difficult and error-prone, and that the indirect approach of the constrained expression formalism is much more straightforward and relatively easy to automate. This process is described in the next section and an example illustrating it is given in section 4 and further elaborated in the appendix. The use of constrained expressions with three very different design notations is demonstrated in [10].

A number of event-based models of concurrent computation have been suggested for the description of software systems. The trace models of Hoare [17] and Misra and Chandy [22] treat system behaviors as sets of sequences of communication events. Axiomatic proof techniques are provided for verifying properties of the systems. In contrast, the constrained expression approach uses a more general concept of event and provides algebraic analysis techniques for establishing properties, such as absence of deadlock and limited use of shared resources, that can be interpreted as questions regarding the order and number of certain event occurrences in behaviors.

Regular expression-like closed form descriptions of the sets of sequences of events representing system behaviors are found in the work of Campbell and Habermann [5] (path expressions), Riddle [29] (message transfer expressions and event expressions), Welter [35] (counter expressions), and Shaw [32] (flow expressions). Message transfer expressions, event expressions, flow expressions, counter expressions and the COSY notation [21], which is based on path expressions, all rely on a filtering procedure, whereby sequences representing impossible behaviors are eliminated from a set containing all legal system behaviors and represented by a single, or possibly several, expressions. The various notations use different, although similar, language operators. The filtering steps are also formulated differently. In the next section, after presenting a more detailed description of constrained expressions, we briefly return to a discussion of these notations and their relationships to constrained expressions.

Greif [12] and Chen and Yeh [6] have developed event-based models that rely on explicit partial orderings of events. A system's behavior is represented by a set of events, along with certain partial order relations. A relation expresses a time order or enabling relationship between events. Events which are not comparable in the partial orders are considered concurrent. As explained above, this model is entirely compatible with the representation of system behaviors as sets of sequences of events.

The constrained expression formalism has a number of advantages that make it an especially appropriate vehicle for adding analysis capabilities to design methods for concurrent software systems. Its major advantages are its generality and broad applicability. Most related approaches, such as those enumerated above, are either inextricably bound to a single, sometimes obscure or idiosyncratic, notation for describing concurrent systems or else exist solely as abstract formalisms with no ties to any congenial, practical notation whatsoever. The constrained expression approach, on the other hand, supports the introduction of an analysis component into a generic paradigm for design of concurrent software systems, as we show in section 4. Its suitability for use in conjunction with a wide variety of practical, real world notations for concurrent systems has been firmly established [9,10]. Its value in performing useful analysis on realistic problems has been demonstrated [3]. Moreover, its independence from any specific programming language or design notation gives the constrained expression formalism the potential to provide uniform, powerful analysis capabilities during not just one but all phases of concurrent software system development.

3. The Constrained Expression Formalism

The constrained expression formalism was first introduced as part of a generalization of the DREAM system [37], which was an early attempt to provide automated support for the design of concurrent systems. DREAM included a design language, called DDN [28], and procedures for deriving *event expressions* representing certain features of the behavior of the system being being designed [39]. These event expressions[1], basically regular expressions with additional operators corresponding to aspects of concurrency, provided closed-form descriptions of the sequences of certain events occurring in behaviors of the system. Special symbols were introduced to express constraints on the order of events having to do with the transmission of messages between parts of the system. Sequences in which pairs of these symbols did not properly correspond were considered to represent illegal behaviors and thus eliminated. The system designer could then compare the intended behavior of the system with the derived event expression description.

DREAM could only describe systems in which the set of constituent processes and the communication paths connecting them remained static. The Dynamic Process Modelling Scheme (DPMS) [38] was developed by Wileden to extend the DREAM approach to systems with dynamic structure. The constrained expression formalism introduced as part of DPMS greatly extended the power and flexibility of the event expressions of DREAM by providing a new mechanism for specifying which strings of event symbols represent illegal behaviors.[2] Dillon [9] subsequently gave a revised formulation of constrained expressions that provides better support for analysis and is easier to use with different design languages. It is this version of the formalism that we now describe.

The constrained expression representation of a distributed system consists of a *system expression* and a collection of *constraints*. The system expression is a regular expression over an alphabet of symbols called the *augmented alphabet*, and can be derived from a description of the system in a suitable notation (such as a design language or a programming language) through the use of a set of translation rules. This augmented alphabet consists of symbols representing events in the system, together with some additional sym-

[1] Event expressions were a descendent of *message transfer expressions* [29].

[2] Constrained expressions were a generalization of counter expressions [35], which in turn were an alternative formulation of DREAM's event expressions.

bols needed to express certain aspects of the semantics of the notation. The usual regular expression operators, concatenation (denoted by juxtaposition), alternation (written $\vee$), and Kleene star (written *), are used in the representation of the activity of the sequential components of a system. As a general rule, concatenation represents the sequencing of events, alternation represents a choice from among a number of different activities (the result, for example, of branches in control flow), and Kleene star represents an arbitrary finite number of iterations of an activity (the result of loops). In order to efficiently express the interleaving of strings representing concurrent activity, we also use a "shuffle" operator, written Δ. Thus, for example, the regular expression $ab \, \Delta \, cd$ represents the set $\{ abcd, acbd, acdb, cabd, cadb, cdab \}$. The shuffle operator has been shown to preserve regularity [11] and is used only as a notational convenience. The relative precedence of these operators, from highest to lowest, is as follows: Kleene star, concatenation, shuffle, and finally, alternation.

The translation rules used to derive a system expression from a description of the system in some design notation capture part of the semantics of that notation. Thus, the prefixes of the regular language associated with the system expression represent event sequences that include all the possible behaviors of the system. (We consider prefixes, rather than complete strings in the language of the system expression, in order to represent behaviors in which parts of the system terminate abnormally.) Some of the semantics of a notation for describing distributed systems are not easily captured by translation rules, partly because of the problem with interleaving event sequences from different processes noted earlier. Thus, some of the prefixes may represent event sequences that cannot be behaviors. The prefixes of the language associated with the system expression, therefore, can be viewed as representing "candidate" event sequences. The constraints of the constrained expression representation are used to eliminate those prefixes that do not represent legitimate system behaviors.

The constraints are also expressions over the augmented alphabet, formed using the regular expression operators and one additional operator, denoted by $\dagger$. The $\dagger$ represents the shuffle of zero or more copies of its argument, and is used in constraints to ensure, for example, that at least as many messages have been sent as have been received. (Note that constraints containing this operator may not be regular expressions.) The constraints define legal patterns of event symbols and embody that part of the semantics of the distributed system development notation not expressed by the translation rules. They are used to "filter" the prefixes of the language of the system expression to eliminate those prefixes that do not correspond to possible system behaviors, in the following manner.

Each constraint is in fact associated with a subalphabet of the augmented alphabet, called a *constraint alphabet*. The expression defining the constraint describes a language which consists of precisely the legal patterns over that subalphabet. We determine whether a prefix of a string in the language of the system expression *satisfies* a constraint by erasing from it all symbols except those in the constraint alphabet and then seeing whether the resulting string is in the language of the constraint. If not, we say that the prefix *violates* the constraint. Repeating this process for each constraint in the constrained expression's *constraint set*, and eliminating any prefix that violates any constraint, reduces the set of prefixes generated from the system expression to a set of *constrained prefixes* that abide by all of the constraints.

The constraints may be thought of as imposing requirements on a sequence of events that must be satisfied if the sequence is to occur in a behavior of the system. Thus, the constrained prefixes correspond to the system behaviors that are actually possible when all aspects of the semantics of the distributed system development notation are taken into account. Symbols representing events that are not of interest when describing the final system behaviors are then erased from the constrained prefixes. The resulting set of strings is the *interpreted language* of the constrained expression. The interpreted language thus represents exactly the possible behaviors of the system. It is important to note that it is never necessary to actually generate this (possibly infinite) language. Analysis techniques operate on the constrained expression itself, rather than on individual strings in the interpreted language.

To make this filtering process clearer, consider a situation in which a process *P* starves waiting for a particular communication to take place. This starvation might be represented by, say, a *starve(P)* symbol. In an actual behavior in which the process starves, no further events involving activity of that process can occur. Thus, we must ensure that there are no constrained prefixes in which a *starve(P)* is followed by a symbol representing activity of the process *P*.

To accomplish this, we include a special "non-event" symbol *ne(P)* in the augmented alphabet and define the system expression so that no symbol representing activity of process P occurs after a *starve(P)* without an intervening *ne(P)*. (The translation rules described in section 4.2 illustrate how this is done.) Then a constraint with alphabet $\{ne(P)\}$ and expression λ (we use λ to represent the empty string) imposes the desired restriction. This is because any prefix containing a *starve(P)* and a symbol representing activity of process P after the starvation must also contain an *ne(P)*, but when we project such a prefix onto the constraint alphabet by erasing all symbols except the *ne(P)*, the result is not in the language of the constraint. Hence, such a prefix violates the constraint. This constraint thus eliminates all prefixes with symbols representing activity of the process after starvation.

The *ne(P)* symbol is an example of the type of symbol added to the augmented alphabet in order to express some aspect of the semantics of the design notation. Other symbols which do not correspond to observable system events may be necessary to express such things as appropriate requirements for synchronization of communication or consistent use of variables. A few of these symbols appear in the example of section 4. These symbols are generally erased from the constrained prefixes as part of the process of forming the interpreted language.

This formulation of constrained expressions, while equivalent to Wileden's original one, uses a "cleaner" and more intuitive set of definitions. In general, it also allows simpler constrained expressions for representing a concurrent system's behaviors. As a result, this version of constrained expressions has been applied to a wider range of languages and phases in concurrent software development than was the original version. The most recent of these is the Ada®-like design language illustrated in the next section. The cleaner and more intuitive definitions also mean that the procedures for translating each of these languages into constrained expressions are more easily automated.

The major difference between the constrained expression formalism and its ancestors (message transfer expressions, event expressions, and counter expressions) is the use of constraints to explicitly specify the conditions under which strings are to be eliminated

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

from the set of possible behaviors described by an expression. The filtering procedures used with the earlier notations are equivalent, in the more general constrained expression formalism, to using a fixed, pre-defined set of constraints expressing synchronization requirements between communicating processes. As mentioned above, for example, special symbols are associated with the transmission of messages in an event expression, and a cancellation rule eliminates strings in which these symbols do not properly correspond. The same effect can be achieved, using constrained expressions, with a single constraint over these special symbols. Additional rules may also be provided in a constrained expression description of a system by constraints over other symbols in the augmented alphabet. Constraints in a constrained expression description of a system, therefore, may express more general requirements than simply those involving synchronization. For instance, as the example above shows, constraints can be provided to eliminate strings in which a process is represented as having starved, and then, at some later point in time, as again participating in some system event.

Like these earlier notations, flow expressions use pre-defined mechanisms for expressing constraints, and thus lack much of the flexibility and generality of constrained expressions. Flow expressions, however, are also distinguished by the use of an additional operator, an infinite repetition operator, to permit the representation of infinite behaviors, and by a different interpretation of events in the resulting expressions. These differences result in differences in descriptive power, ease of use for both description and analysis, and range of application [32,33].

The COSY notation [21] closely resembles constrained expressions (although the dramatically different syntax of these notations obscures their underlying similarity). Differences in the two notations are primarily the result of a difference in their intended use. COSY was intended to be used directly by software developers for specifying concurrent software systems, not indirectly through a translation process like constrained expressions. The events in a COSY description, therefore, correspond to operations or procedures, rather than arbitrary system events. Path expressions, the COSY analogue of constraints, specify the order in which operations can be invoked. A COSY description thus provides a non-algorithmic specification of the intended system behaviors, presumably for use in verification of a design or implementation. A constrained expression, on the other hand, is not taken as defining the intended observable behaviors of a system, but is used for exploring properties of a particular design. Moreover, of course, the constrained expression formalism permits the use of a wide variety of notations, while COSY, itself intended for expressing specifications, admits no such generality.

3.1. Algebraic Analysis of Constrained Expression Descriptions

The constrained expression formalism supports a useful, general-purpose technique, based on elementary algebra, for analyzing event-based descriptions of the behavior of concurrent systems [2,3]. This technique provides a basis for arguments concerning the order and number of events that occur in the behaviors of a system. Such arguments play an important role in the analysis of distributed systems, and they have been widely, if informally, applied (see, for example, [5,13,18,19,20,22]). Behavioral properties such as mutual exclusion, deadlock, and starvation, which involve interactions among the parts of a distributed system, are most naturally analysed in terms of the order and number of event occurrences. Because a constrained expression description of a system explicitly encodes the order and number of event symbols in sequences representing the behaviors of the system, it is well-suited for use in arguments of this nature.

The fundamental approach of the algebraic analysis techniques is to determine whether a particular event, or (sub)sequence of events, appears in any possible system behavior. This can be viewed as a generalization of the technique employed by Habermann in analysing a semaphore solution to a producer-consumer problem [13]. In the case of a constrained expression representation, this approach corresponds to asking whether some particular event symbol, or pattern of event symbols, occurs in any string described by the constrained expression.

The analysis technique is based on iterative generation of inequalities that characterise properties of a system's possible behaviors. These inequalities are generated in the following way. Certain fundamental properties of any event-based model of computation (many of which are expressed by constraints in a constrained expression formulation of the model) can be regarded as conditions on the collection of events preceding the occurrence of a given event. In terms of behavior strings, these conditions produce a set of inequalities involving the numbers of occurrences of various symbols in the segment of the string preceding a given symbol. To determine whether there is a behavior containing a specified pattern of events, we begin by assuming that these events do occur in the string, then generate the inequalities for the segments that would precede them. These relations in turn involve occurrences of other symbols, and we generate new inequalities on the segments preceding these. Continuing in this fashion, we attempt to determine whether the inequalities are inconsistent, in which case no behavior contains the specified pattern. If the relations are consistent, we use them in an attempt to produce a behavior containing the pattern. This focused approach reduces the combinatorial problems incurred by exhaustive analysis of all possible system behaviors. We illustrate the approach in the following section and the appendix.

4. Constrained Expressions and System Designs

In this section we show how constrained expressions can be used during the design phase of software development. In general, our approach is to use the constrained expression formalism to augment any of a variety of concurrent system design methods by adding important analysis capabilities. The augmented design method can then be viewed as progressing through the following stages, which we describe and illustrate below.

- design formulation
- constrained expression generation
- analysis

Our approach permits the designer to formulate the design in a congenial notation. The design is then mechanically translated into a constrained expression which describes exactly the possible behaviors of the system. The constrained expression is subsequently analyzed to determine whether the system behaves as intended.

This approach permits a developer to determine if a design is satisfactory before using it to produce a more elaborate design or an implementation. (Of course, the analysis may be performed by someone other than the original system designer.) It may in fact be necessary to modify the design to correct some problem revealed by analysis. A design is thus used as the basis for further system development only after it has been determined to display the intended behavior. The above three stages, therefore, are repeated, first, as the design is modified, and then again, as the design is elaborated. In practice, the analysis results obtained using an early design can often be used with a later design, so that unnecessary duplication of effort is avoided.

4.1. Design formulation

To illustrate the approach, we consider Dijkstra's dining philosophers problem. This problem has been widely studied because, despite its relative simplicity, a solution requires that the following questions be addressed.

(1) Are resources in the system used in a mutually exclusive fashion?

(2) Can the system deadlock?

These questions are typical of the types of questions that concern designers of concurrent systems.

The dining philosophers problem can be described as follows. Five philosophers share a common dining room containing a circular table surrounded by five chairs and set with five forks. Each philosopher requires two forks in order to eat dinner, which is always a tangled mess of spaghetti. The only forks available to a philosopher are those on the immediate right and left of his/her place. The problem is to design a system that models the activities of the five philosophers as they periodically think and then eat.

As a first step, a developer must come up with a preliminary design for the system under development. This design could be developed using any of a variety of methods and expressed in any design notation for which a translation procedure into constrained expressions has been formulated. A design focusing on interprocess communication and synchronization, for instance, might be written in DDN. For the example in this section, we use an Ada-like notation.

Our initial design contains ten processes, which are described as *tasks* in the Ada-like notation. Five tasks, P_0, P_1, P_2, P_3, and P_4, model the behavior of the five philosophers, and five tasks, F_0, F_1, F_2, F_3, and F_4, corresponding to the five forks on the table, represent the philosophers' use of the forks. The designs for these tasks are shown in Figure 1. Two differences between Ada and the design notation used here are apparent in these designs. First, the boolean expression *internal_test* in the while loop of task P_i is to be evaluated nondeterministically. This special expression is used in the design notation to model some as yet unspecified computation. The second difference is the use of ellipses in place of the bodies of the procedures Think and Eat, leaving the computations performed by these procedures unspecified. (Arithmetic in this figure and the rest of the discussion pertaining to this example is performed modulo 5.)

4.2. Constrained expression generation

Given the design for a system, a constrained expression whose interpreted language describes all the possible behaviors of the system is then produced. This is done in two steps. A constrained expression is first *derived* from the design. This is essentially a compilation step, being completely algorithmic and based on a set of translation rules, and produces an appropriate internal form for subsequent simplification and analysis. The derived constrained expression is then "simplified", yielding a constrained expression with the same interpreted language. This simplified constrained expression therefore describes the same set of behaviors as the derived constrained expression, but is easier to analyze. The simplification of the derived constrained expression is analogous to an optimization phase following compilation. This two step approach to producing constrained expressions from designs allows us to associate fairly straightforward derivation procedures with the various design notations without regard for efficiency of analysis. Once a constrained expression description for a system has been produced, general simplification procedures can be applied to facilitate subsequent analysis. Here we illustrate part of this process for producing a constrained expression from the design of Figure 1. Simplification procedures and analysis are discussed more completely in the appendix.

As described in section 3, the constrained expression representation of a system has two components: a system expression and a set of constraints. The system expression represents the unconstrained activity of the individual components of the system. The constraints impose appropriate restrictions on their activity. For example, constraints are used to assure that two tasks are synchronized when communicating by rendezvous, and that, if a task starves waiting to accept a call on one of its entries, that task does not participate in any subsequent system events.

The system expression for the design presented above is obtained by interleaving regular expressions, called *task expressions*, describing the sequential activity of the tasks in the system. The task expressions are obtained from the designs of the tasks through

```
task F_i is
    entry U;
    entry D;
end F_i;

task body F_i is
begin
    loop
        accept U;   --fork is picked up
        accept D;   --fork is put down
    end loop;
end F_i;
```

```
task P_i;

task body P_i is
begin
    while internal_test loop
        Think;
        F_{i+1}.U;   --left fork up
        F_i.U;       --right fork up
        Eat;
        F_i.D;       --right fork down
        F_{i+1}.D;   --left fork down
    end loop;
end P_i;
```

```
procedure Think is ..;

procedure Eat is ..;
```

Figure 1

Designs for the fork task F_i, the philosopher task P_i

and the Think and Eat procedures

the statement-by-statement application of a set of translation rules. The translation rules are determined by the semantics of the design notation. (Thus, different translation rules are required for different design notations. Examples of translation rules for three very different concurrent systems descriptive notations appear in [10].)

The translations of the statements "accept U" and "F_i.U", appearing, respectively, in the designs of the tasks F_i and P_i, are shown in Figure 2, where the event symbols used in this figure are defined in Figure 3. Given these definitions, the translation rules are easy to understand. The "accept U" statement, for example, translates into three alternatives: the first two represent the possibility that execution of this statement eventually results in a rendezvous with one of the tasks P_i or P_{i-1} (the only tasks that call the entry F_i.U) and the third represents the possibility that no task ever calls F_i.U after this point (as described earlier, the non-event symbol $ne(F_i)$ is used in a constraint to assure that there are no constrained prefixes in which a $starve_a(F_i.U)$ symbol is followed by a symbol representing an activity of the task F_i.)

Applying the appropriate translation rules to the designs of the tasks F_i and P_i, for $0 \leq i \leq 4$, produces the task expressions shown in Figure 4. Given the interpretation of the event symbols shown in Figure 3, it is apparent that the task expressions represent the sequential activity of the individual tasks in the system.

The task expressions are interleaved when forming the system expression in order to represent the concurrency in the system. This produces a regular expression whose language of prefixes contains a string representing each possible behavior of the system. As pointed out in section 2, however, some interleavings of strings from the languages of the task expressions may produce prefixes representing event sequences that violate the semantics of the design notation. Thus, for example, the *resume* symbol generated by the translation of the first call statement in the design of the task P_1 can be interleaved before the *beg_rend* symbol generated by the translation of the corresponding accept statement in the body of the task F_2, as in the prefix

$$inv(P_1;\text{Think})\,call(P_1;F_2.U)\,resume(P_1;F_2.U)\,beg\_rend(P_1;F_2.U)\,end\_rend(P_1;F_2.U).$$

The constraints are used to filter out any such prefixes that do not represent possible behaviors of the system.

- **in the body of task F_i:**

  accept U ::=

  $$\mathit{beg\_rend}(P_i; F_i.U)\,\mathit{end\_rend}(P_i; F_i.U)$$
  $$\lor\ \mathit{beg\_rend}(P_{i-1}; F_i.U)\,\mathit{end\_rend}(P_{i-1}; F_i.U)$$
  $$\lor\ \mathit{starve}_a(F_i.U)\,\mathit{ne}(F_i)$$

- **in the body of task P_i:**

  F_i.U ::=

  $$\mathit{call}(P_i; F_i.U)\,\mathit{resume}(P_i; F_i.U) \lor \mathit{starve}_c(P_i; F_i.U)\,\mathit{ne}(P_i)$$

Figure 2

Sample translation rules

Symbol	*Associated Event*
$call(T1;T2.E)$	Task $T1$ calls $T2.E$
$resume(T1;T2.E)$	Execution of task $T1$ can resume after rendezvous on $T2.E$
$starve_c(T1;T2.E)$	Task $T1$ starves waiting for call on $T2.E$ to be accepted
$beg\_rend(T1;T2.E)$	Begin rendezvous between tasks $T1$ and $T2$ on $T2.E$
$end\_rend(T1;T2.E)$	End rendezvous between tasks $T1$ and $T2$ on $T2.E$
$starve_a(T1.E)$	Task $T1$ starves waiting to accept a call on $T1.E$
$inv(T1;P)$	Task $T1$ invokes the procedure P
$beg\_loop(T1)$	Task $T1$ begins execution of loop statement
$end\_loop(T1)$	Task $T1$ completes execution of loop statement
$comp(T1)$	Task T1 completes execution
$ne(T1)$	Non-event symbol for task T1

Figure 3

Some event symbols and associated events

Task expression for F_i:

$$\Big[\Big(\textit{beg\_rend}(P_i; F_i.U)\,\textit{end\_rend}(P_i; F_i.U) \vee \textit{beg\_rend}(P_{i-1}; F_i.U)\,\textit{end\_rend}(P_{i-1}; F_i.U)$$
$$\vee\ \textit{starve}_a(F_i.U)\,\textit{ne}(F_i)\Big)$$
$$\Big(\textit{beg\_rend}(P_i; F_i.D)\,\textit{end\_rend}(P_i; F_i.D) \vee \textit{beg\_rend}(P_{i-1}; F_i.D)\,\textit{end\_rend}(P_{i-1}; F_i.D)$$
$$\vee\ \textit{starve}_a(F_i.D)\,\textit{ne}(F_i)\Big)\Big]^* \textit{ne}(F_i)\,\textit{comp}(F_i)$$

Task expression for P_i:

$$\Big[\textit{inv}(P_i; \text{Think})\Big(\textit{call}(P_i; F_{i+1}.U)\,\textit{resume}(P_i; F_{i+1}.U) \vee \textit{starve}_c(P_i; F_{i+1}.U)\,\textit{ne}(P_i)\Big)$$
$$\Big(\textit{call}(P_i; F_i.U)\,\textit{resume}(P_i; F_i.U) \vee \textit{starve}_c(P_i; F_i.U)\,\textit{ne}(P_i)\Big)\,\textit{inv}(P_i; \text{Eat})$$
$$\Big(\textit{call}(P_i; F_i.D)\,\textit{resume}(P_i; F_i.D) \vee \textit{starve}_c(P_i; F_i.D)\,\textit{ne}(P_i)\Big)$$
$$\Big(\textit{call}(P_i; F_{i+1}.D)\,\textit{resume}(P_i; F_{i+1}.D) \vee \textit{starve}_c(P_i; F_{i+1}.D)\,\textit{ne}(P_i)\Big)\Big]^* \textit{comp}(P_i)$$

Figure 4

Task expressions derived from the designs of F_i and P_i

The constraints express that part of the semantics of the design notation not captured by the translation rules. For a particular design notation we identify a set of constraint templates which, when instantiated for any particular design, produces the set of constraints required for that design. For example, a single template is used to ensure that tasks are properly synchronized during a rendezvous. Instantiating this template for the task P_1 and for the entry $F_2.U$, we obtain the constraint,

$$\left(\mathit{call}(P_1; F_2.U)\,\mathit{beg\_rend}(P_1; F_2.U)\,\mathit{end\_rend}(P_1; F_2.U)\,\mathit{resume}(P_1; F_2.U)\right)^*,$$

describing the legal patterns of $\mathit{call}(P_1; F_2.U)$, $\mathit{beg\_rend}(P_1; F_2.U)$, $\mathit{end\_rend}(P_1; F_2.U)$, and $\mathit{resume}(P_1; F_2.U)$ symbols that can appear in strings representing behaviors of the system.[1] Note that this constraint would prevent the prefix containing the premature $\mathit{resume}(P_1; F_2.U)$, cited in the preceding example, from surviving the filtering process and being included in the set of constrained prefixes. Note also that while this constraint forces a "call" event to always precede the corresponding "begin rendezvous" event, it does not impose any ordering on the occurrence of an entry call and the corresponding accept. (In a design notation in which this order was significant, additional symbols would be included in the constrained expression representation to make those events visible.) Similarly, although this constraint requires that an $\mathit{end\_rend}(P_1; F_2.U)$ precede the corresponding $\mathit{resume}(P_1; F_2.U)$, it does not stipulate whether the calling task or the accepting task will be the first to actually execute an instruction following the completion of the rendezvous.

In general, we have kept the derivation procedure as transparent as possible by using simple and understandable translation rules and constraints. Consequently, the expression obtained is often not as simple as it could be. The next step in producing a constrained expression representation for a system, therefore, is to take advantage of specific features of the system to simplify the expression derived from the design. Thus, for example, certain symbols and constraints may be superfluous, due to the form of the system expression and other constraints. These would be eliminated. Individual alternatives within the system expression or a constraint might also be eliminated. Many of these simplifications are routine and can be done automatically, in the same way that certain optimizations can be

[1] This is the constraint that is used with a simplified version of the design notation which, for example, does not permit nested rendezvous.

done automatically as part of a compilation. Part of the routine simplification process for the constrained expression derived for the example of figure 1 is illustrated in the appendix.

Other simplifications use the *message flow analysis algorithms* [9]. These algorithms take into account the manner in which data flows between the tasks in the system to simplify the task expressions. They resemble standard data flow analysis algorithms and, once the designer has identified a set of entries to monitor, can be applied in a purely mechanical fashion.

4.3. Analysis

The fundamental approach to constrained expression-based analysis of designs is to determine whether a particular symbol or pattern of symbols appears in a string in the interpreted language of the constrained expression. The symbols in question may correspond to some desirable property of the system, such as mutually exclusive use of some shared resource, or graceful degradation and continued operation following the failure of one or more system components. Alternatively, they might represent pathological behaviors such as deadlocks. For example, to determine if the philosopher tasks are appropriately synchronized in the above design (i.e., if the philosophers use the forks in a mutually exclusive fashion), one could ask if there are any strings in the interpreted language of a constrained expression representation for the system containing *beg_rend*$(P_i; F_i.U)$ and *beg_rend*$(P_{i-1}; F_i.U)$ symbols (representing adjacent philosophers picking up the same fork) with no intervening *end_rend*$(P_i; F_i.D)$ or *end_rend*$(P_{i-1}; F_i.D)$ symbols (representing a philosopher putting down the fork). Similarly, the situation in which a philosopher becomes permanently unable to pick up a fork is represented by a string in the interpreted language containing a *starve*$_c(P_i; F_i.U)$ or a *starve*$_c(P_i; F_{i+1}.U)$ symbol.

A central technique for the analysis of constrained expressions is based on the algebraic methods described in section 3.1. A particular pattern of event symbols is assumed to appear in a string from the interpreted language of the constrained expression. Of course this pattern must also appear in a constrained prefix of the constrained expression. The system expression and the constraints are then used to generate inequalities involving the numbers of occurrences of particular event symbols in various segments of the hypothesized constrained prefix.

For example, to determine if a philosopher, as modeled in the above design, can starve trying to pick up his/her right fork, we need to determine if a $\mathit{starve}_c(\mathrm{P}_i; \mathrm{F}_i.\mathrm{U})$ symbol appears in any string representing a behavior of the system. We therefore assume that there is some constrained prefix containing a $\mathit{starve}_c(\mathrm{P}_i; \mathrm{F}_i.\mathrm{U})$ symbol and examine the system expression and various constraints to produce inequalities relating the number of occurrences of different symbols in this constrained prefix. One of the constraints relating to the $\mathit{starve}_c(\mathrm{P}_i; \mathrm{F}_i.\mathrm{U})$ symbol, for instance, implies that there are no $\mathit{starve}_a(\mathrm{F}_i.\mathrm{U})$ symbols in the hypothesized constrained prefix. (This constraint is given in the appendix.) Examination of the system expression in a simplified constrained expression representation for the system (also given in the appendix) then shows that the number of $\mathit{beg\_rend}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{U})$ symbols in this constrained prefix is one more than the number of $\mathit{beg\_rend}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{D})$ symbols. Additionally, the synchronization constraints described above imply that the number of $\mathit{beg\_rend}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{U})$ symbols is equal to the number of $\mathit{call}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{U})$ symbols, while the number of $\mathit{beg\_rend}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{D})$ symbols is equal to the number of $\mathit{call}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{D})$ symbols. It must be the case, therefore, that the number of $\mathit{call}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{U})$ symbols in the hypothesized constrained prefix is one more than the number of $\mathit{call}(\mathrm{P}_{i-1}; \mathrm{F}_i.\mathrm{D})$ symbols. But examination of the system expression in the simplified constrained expression reveals that for this latter equality to hold the constrained prefix must also contain a $\mathit{starve}_c(\mathrm{P}_{i-1}; \mathrm{F}_{i-1}.\mathrm{U})$ symbol. Hence a constrained prefix contains a $\mathit{starve}_c(\mathrm{P}_i; \mathrm{F}_i.\mathrm{U})$ symbol only if it also contains a $\mathit{starve}_c(\mathrm{P}_{i-1}; \mathrm{F}_{i-1}.\mathrm{U})$ symbol. In terms of the modeled system, this means that a philosopher can starve trying to pick up his/her right fork only if the philosopher on his/her right does also. All the philosophers must therefore starve trying to pick up their right forks. The above analysis is described more rigorously in the appendix.

Given the information revealed by this analysis, it is not difficult to generate the constrained prefix shown in figure 5. Clearly, this constrained prefix corresponds to a behavior in which the philosophers all first think, then pick up the forks on their left sides, and finally starve trying to pick up the forks on their right sides.

For the analysis above, it was sufficient to consider the number of event occurrences in the hypothesized constrained prefix. To establish certain other properties, however, it is also necessary to determine the order of various event occurrences in a hypothesized con-

$$\mathit{inv}(P_0;\text{Think})\,\mathit{inv}(P_1;\text{Think})\,\mathit{inv}(P_2;\text{Think})\,\mathit{inv}(P_3;\text{Think})\,\mathit{inv}(P_4;\text{Think})$$

$$\mathit{call}(P_0;F_1.U)\,\mathit{call}(P_1;F_2.U)\,\mathit{call}(P_2;F_3.U)\,\mathit{call}(P_3;F_4.U)\,\mathit{call}(P_4;F_0.U)$$

$$\mathit{beg\_rend}(P_0;F_1.U)\,\mathit{beg\_rend}(P_1;F_2.U)\,\mathit{beg\_rend}(P_2;F_3.U)\,\mathit{beg\_rend}(P_3;F_4.U)\,\mathit{beg\_rend}(P_4;F_0.U)$$

$$\mathit{end\_rend}(P_0;F_1.U)\,\mathit{end\_rend}(P_1;F_2.U)\,\mathit{end\_rend}(P_2;F_3.U)\,\mathit{end\_rend}(P_3;F_4.U)\,\mathit{end\_rend}(P_4;F_0.U)$$

$$\mathit{resume}(P_0;F_1.U)\,\mathit{resume}(P_1;F_2.U)\,\mathit{resume}(P_2;F_3.U)\,\mathit{resume}(P_3;F_4.U)\,\mathit{resume}(P_4;F_0.U)$$

$$\mathit{starve}_c(P_0;F_0.U)\,\mathit{starve}_c(P_1;F_1.U)\,\mathit{starve}_c(P_2;F_2.U)\,\mathit{starve}_c(P_3;F_3.U)\,\mathit{starve}_c(P_4;F_4.U)$$

$$\mathit{starve}_a(F_0.D)\,\mathit{starve}_a(F_1.D)\,\mathit{starve}_a(F_2.D)\,\mathit{starve}_a(F_3.D)\,\mathit{starve}_a(F_4.D)$$

Figure 5

A constrained prefix demonstrating the potential for deadlock

strained prefix. To show that the forks, as modeled in the design above, are used properly, for example, it is necessary to determine how symbols from different task expressions (i.e., the symbols associated with a philosopher picking up a fork and the symbols associated with a philosopher putting down a fork) are interleaved to form constrained prefixes. This is accomplished by using the system expression and constraints to reason about the number of occurrences of certain event symbols in various initial segments of a hypothesized constrained prefix.

After analyzing the design presented in Figure 1, a developer might modify it to eliminate the potential for deadlock revealed by the analysis. For instance, a task can be introduced to limit the number of philosopher tasks within their critical sections (i.e., between the calls to the U and D entries of the left-hand fork task) at any point during a behavior to four. (This models the solution suggested in [16], whereby an attendant guards the entrance to the dining room and never allows more than four philosophers to be in the room at the same time.)

Analysis of this new design would need to address the same questions as analysis of the original design. If certain relationships between the constrained expression representations of the original and modified designs are maintained, however, there is no need to repeat the analysis performed with the original design, as it can be shown to apply equally well with the modified constrained expression. This approach to the modularization of constrained expression-based analysis is illustrated in [9].

Of course, analysis of the modified design would also need to address some additional questions regarding the behavior of the modified system. The newly introduced task, for example, might maintain a variable whose value is intended to indicate the number of philosopher tasks within their critical sections at any point during a behavior. In this case, we use the sort of reasoning described above to show that the desired relationship between the value of this variable and the state of the philosopher tasks is actually realized and that the value of this variable is never greater than four. This information, together with the characterization of deadlocking behaviors obtained earlier, then assures that the system is free from deadlock.

5. Experience

A variety of experience with the constrained expression approach and related event-based methods for designing concurrent systems reinforces our belief that this approach can be of significant value to developers of concurrent software. In this section we provide a brief overview of that experience.

An early version of event-based design was explored in [30]. This version of the approach, tailored for use with DREAM, focused primarily on the description of a concurrent system's organization and included few of the analysis capabilities found in the later constrained expression formulation. Nevertheless, several experiments with this early version [8,27,31,34,36] demonstrated that this approach was well-suited for application to concurrent software development problems. In particular, its orientation toward high-level description of process activities and process interactions, as provided through DDN, proved to be extremely natural for the formulation of a wide range of concurrent system designs.

Our experience with analysis techniques associated with the constrained expression formalism is also encouraging. Such analysis based on the number and order of event occurrences is an apt and powerful tool for assessing designs of concurrent systems. A good example of the merits of this approach appears in [3]. In that paper a complex design for a solution to the distributed mutual exclusion problem, due to Ricart and Agrawala [25], was subjected to the algebraic analysis techniques outlined in section 3.1. A subtle flaw in the originally published version of the design (which Ricart and Agrawala had corrected in [26]) was rapidly pinpointed by the algebraic analysis techniques. When that flaw had been repaired, the techniques permitted rigorous proof that the corrected version of the design exhibited precisely the intended behavior.

The generality of the constrained expression approach has also been convincingly demonstrated. In Dillon's thesis [9], it is employed in conjunction with several quite different design languages for concurrent systems. One of those languages is a dialect of DYMOL [38], a descendent of DDN that utilizes asynchronous message transmission as its interprocess communication medium. Dillon [9] developed detailed translation rules and analysis techniques for the constrained expressions corresponding to this DYMOL dialect and also a derivation procedure for a dialect of CSP [16]. Like the Ada-based design language used in the examples of section 4, this CSP dialect uses synchronous interchange as its interprocess communication medium. Dillon [9] also gave translation rules for use with Petri nets [24], the classic model of concurrent computation which has been extensively investigated during the past 25 years. In this application, the constrained expression formalism provided a novel, closed form representation for Petri net languages [14,23].

Finally, our experience with automating this approach has been encouraging. A partial prototype implementation of the DREAM system provided the first experience with tools supporting this approach. More recently, a prototype tool for translating DYMOL into constrained expressions has been implemented [15] and a simulator for DYMOL has been built [40]. Exploration of tool support for the analysis techniques associated with event-based design has led to a prototype tool for event sequence generation from a constrained expression representation [1]. Although other analysis tools will be more complex, requiring more powerful reasoning and heuristic capabilities than the prototype tools developed to date, this experience with automated support for the approach substantiates our belief that it is amenable to automation. Our future plans along these lines are described in the next section.

The focus of our work on the constrained expression approach has been on establishing its practical utility and generality. Thus, rather than determining the formal, theoretical properties and limitations of our description and analysis techniques, we have concentrated on applying them to different types of design notations and experimenting with their capabilities for establishing important synchronization properties of concurrent system designs. Initially, we have intentionally avoided language constructs that might unduly complicate the derivation and analysis techniques. The translation rules and constraint templates used with the Ada-like designs described above, for example, rely on certain simplifying assumptions regarding the nature of these designs. Most notably, we require that the tasks in a system are neither created nor destroyed during execution, but are known statically and activated simultaneously. Similarly, a limited set of primitive types are assumed (essentially enumeration types, which are easily represented and are appropriate for the high-level expression of control flow dependencies). The extent to which these and other restrictions can be relaxed and their effect on analysis is a topic for future research. Experience indicates, however, that language features requiring dynamic identification, such as access types and subscripts, will present many of the same sorts of problems for constrained expression-based analysis as they do for symbolic execution.

6. Conclusions and Future Directions

Our highest priority for continued development of our approach is implementation of additional and improved tools to support its use. Specifically, we believe that an appropriate toolset would consist of a constrained expression *deriver*, a *behavior generator*, a constrained expression *simplifier* and an *analyzer*.

The constrained expression deriver is a tool for creating a constrained expression representation corresponding to a description expressed in some design language. Our immediate goal is to complete a deriver specifically applicable to our Ada-based design notation. A table-driven implementation, however, will make the deriver easily adaptable for application to other design languages.

The behavior generator is a tool for producing example behaviors from a constrained expression representation. As mentioned above, a simple prototype behavior generator has already been implemented. This version is capable of producing an arbitrary element of the language of behaviors described by a given constrained expression, and can also be interactively guided in a search for a behavior possessing specific properties. The capabilities offered by a behavior generator are extremely important for effective use of the event-based approach. First, it allows a designer to interactively explore properties of the system. Such exploration can provide important insights, leading to improvements in the design, as well as allowing the immediate detection of certain kinds of errors. Second, and perhaps even more important, when our algebraic analysis technique is unable to determine that a particular pattern of events does not occur in behaviors of a designed system (because the system of inequalities is consistent), the analysis produces a great deal of information about the behaviors containing that pattern. The system developer must then use this information to try to produce an example behavior containing the pattern in order to further understand how it can arise. For systems of realistic size, this can involve a vast amount of bookkeeping, and the use of a tool such as the generator will be essential.

The simplifier will significantly enhance the usefulness and performance of the toolset. This tool will perform simplifications on constrained expressions, with an effect similar to that achieved by simplifying algebraic expressions. After simplification, it will be substantially easier to determine and reason about the possible behaviors represented by a constrained expression. It will often be possible to see from the form of the simplified constrained expression that entire classes of behaviors are impossible. A variety of constrained expression simplification techniques, many of them quite straightforward to implement, are described in [9]. Some of these techniques and the manner in which they are used to reduce the set of possible behaviors are illustrated in the appendix.

The constrained expression analyzer is a generic name for a collection of tools implementing specific analysis techniques applicable to constrained expressions. The tools composing the analyzer will benefit from the constrained expression simplifications performed by the simplifier. While some aspects of the analyzer can be implemented quite directly, others require further research. In particular, appropriate methods and heuristics must be developed for guiding the application of the rules for generating inequalities that are the basis of the algebraic technique. It is clear, however, that even partially automated support for constrained expression analysis will be an extremely valuable aid for developers of concurrent software systems.

Another of our goals is to extend the constrained expression approach to encompass other aspects of the software development process beyond design. We believe, for example, that the constrained expression approach would be very well-suited for specifications of concurrent software systems. Evidence for its broad applicability also comes from two other projects currently underway at the University of Massachusetts. In one of these, an event-based language closely related to the constrained expression formalism is being used as a basis for high-level debugging of concurrent systems [4]. A prototype toolset supporting this debugging method has been implemented and is currently being used in conjunction with a distributed problem-solving testbed system by a research group at the University. Essentially the same event-based language is being used by another research group in an intelligent user interface system that is part of an office automation project [7]. Here the language is used to describe various office procedures to the interface system, which then uses those descriptions to guide or assist users in carrying out the office procedures. We plan to eventually incorporate this interface technology into our own event-based design toolset.

Naturally, we also intend to continue expanding and improving our repertoire of constrained expression design capabilities and analysis techniques. For example, early work on producing a deriver for our Ada-based design notation has resulted in a natural extension of the constrained expression formalism. The extended formalism was produced to permit the description of the FIFO ordering of tasks in the queues associated with entries, which we were not able to characterize using constraints describing permissible patterns of event symbols. The extended formalism also allows more efficient representation of certain other constraints on system behaviors and is compatible with existing constrained expression-based analysis techniques. Another area of particular interest is the application of the constrained expression approach to the design of real time concurrent systems. We believe that the notions of events and sequences of events should be particularly natural for designers of systems that are primarily concerned with the timing and ordering of event occurrences.

In sum, our research on and experience with event-based methods for concurrent software design, especially as realized in the constrained expressions approach, encourage us to believe that our approach can be of significant value to developers of concurrent systems. The constrained expression formalism provides a rigorous conceptual model for the semantics of concurrent computations, which our approach exploits to produce rigorous analysis of important system properties as part of the design process. At the same time, it allows developers to use standard specification and design languages, rather than forcing them to deal with the formal model explicitly or directly. As a result, our approach offers developers the benefits of formal rigor without the associated pain of using unnatural concepts or notations to generate their designs.

7. Acknowledgments

We appreciate the helpful comments of the editor and referees on an earlier draft of this paper. George Avrunin's work has been supported in part by a Faculty Research Grant from the University of Massachusetts. Laura Dillon's work has been supported in part by the IBM Graduate Fellowship Program.

References

[1] S. Avery, "Development of a Behavior Generator for Constrained Expressions," Dept. of Comp. and Info. Science, Univ. of Massachusetts, Amherst, SDLM/84-2, June 1984.

[2] G. Avrunin and J. Wileden, "Algebraic Techniques for the Analysis of Concurrent Systems," *Proc. Sixteenth Annual Hawaii International Conference on System Sciences*, 51–57, 1983.

[3] G. Avrunin and J. Wileden, "Describing and Analyzing Distributed System Designs," *ACM Transactions on Programming Languages and Systems*, to appear.

[4] P. Bates and J. Wileden, "High Level Debugging of Distributed Systems," *Journal of Systems and Software*, vol. 3, no. 4, 255–264, December 1983.

[5] R. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, Heidelberg, 1974, 89–102.

[6] B. Chen and R. T. Yeh, "Formal Specification and Verification of Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-9, no. 6, 710–722, November 1983.

[7] W. Croft and L. Lefkowitz, "Task Support in an Office System," *ACM Trans. on Office Info. Systems*, vol. 2, no. 3, 197–212, July 1984.

[8] J. Cuny, "A Dream Model of the RC4000 Multiprogramming System," Dept. of Comp. and Comm. Sci., Univ. of Michigan, Ann Arbor, RSSM/48, July 1977.

[9] L. Dillon, "Analysis of Distributed Systems using Constrained Expressions," Ph.D. Dissertation, Dept. of Comp. and Info. Science, University of Massachusetts, Amherst. Available as Dept. of Computer and Info. Science Technical Report TR 84-18, September 1984.

[10] L. Dillon, G. Avrunin and J. Wileden, "Constrained Expressions: A General Technique for Describing Behavior of Concurrent Systems", Dept. of Comp. and Info. Science, University of Massachusetts, Amherst, Technical Report TR 85-17, July 1985.

[11] S. Ginsburg, *The Mathematical Theory of Context-Free Languages*. McGraw Hill, New York, 1966.

[12] I. Greif, "A Language for Formal Problem Specification," *Communications of the ACM*, vol. 20, no. 12, 931–935, December 1977.

[13] A. Habermann, "Synchronization of Communicating Processes," *Communications of the ACM*, vol. 25, no. 3, 171–176, March 1972.

[14] M. Hack, "Petri Net Languages," Computation Structures Group, Massachusetts Institute of Technology, Cambridge, Memo 124, June 1975.

[15] P. Ho, "A DC DYMOL to DC Constrained Expressions Translator," Master's Thesis, Dept. of Comp. and Info. Science, Univ. of Massachusetts, Amherst, November 1979.

[16] C. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, 666–677, August 1978.

[17] C. Hoare, "A Calculus of Total Correctness for Communicating Processes," Oxford University Computing Laboratory, Programming Research Group, Oxford, England, Technical Monograph PRG-23, April 1981.

[18] C. Hoare, "Specifications, Programs and Implementations," Oxford University Computing Laboratory, Programming Research Group, Oxford, England, Technical Monograph PRG-29, June 1982.

[19] G. Holzmann, "A Theory for Protocol Validation," *IEEE Trans. on Computers*, 730–738, August 1982.

[20] L. Lamport, "A New Approach to Proving the Correctness of Multiprocess Programs," *ACM Trans. on Programming Languages and Systems*, 84–97, July 1979.

[21] P. Lauer, P. Torrigiani and M. Shields, " COSY: A System Specification Language Based on Paths and Processes," *Acta Informatica*, 451–503, 1979.

[22] J. Misra and K. Chandy, "Proofs of Networks of Processes," *IEEE Trans. on Software Engineering*, vol. SE-7, no. 4, 417–426, July 1981.

[23] J. Peterson, "Computation Sequence Sets," *Journal of Comp. and System Sciences*, vol. 13, no. 1, 1–24, August 1976.

[24] J. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, 223–252, September 1977.

[25] G. Ricart and A. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, vol. 24, no. 1, 9–17, January 1981.

[26] G. Ricart and A. Agrawala, Corrigendum *Communications of the ACM*, vol. 24, no. 9, p. 578, September 1981.

[27] W. Riddle, "DREAM Design Notation Example: The T.H.E. Operating System," Dept. of Comp. Sci., Univ. of Colorado, Boulder, RSSM/50, April 1978.

[28] W. Riddle, J. Wileden, J. Sayler, A. Segal and A. Stavely, "Behavior Modeling during Software Design," *IEEE Trans. on Software Engineering*, vol. 4, no. 4, 283–292, July 1978.

[29] W. Riddle, "An Approach to Software System Behavior Modeling," *Computer Languages*, Vol. 4, 29–47, 1979.

[30] W. Riddle, "An Event-based Design Methodology Supported by DREAM," in Schneider (ed.), *Formal models and Practical Tools for Information Systems Design*. North-Holland Pub. Co., Amsterdam, 1979.

[31] A. Segal, "DREAM Design Notation Example: A Multiprocessor Supervisor," Dept. of Comp. and Comm. Sci., Univ. of Michigan, Ann Arbor, RSSM/53, August 1977.

[32] A. Shaw, "Software Descriptions with Flow Epressions," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, 242-254, May 1978.

[33] A. Shaw, "Software Specification Languages based on Regular Expressions," Institut für Informatik, Eidgenössische Technische Hochschule, Zürich, June 1979.

[34] A. Stavely, "DREAM Design Notation Example: An Aircraft Engine Monitoring System," Dept. of Comp. and Comm. Sci., Univ. of Michigan, Ann Arbor, RSSM/49, July 1977.

[35] M. Welter, "Counter Expressions," Dept. of Comp. Science, Univ. of Michigan, Ann Arbor, RSSM/24, October 1976.

[36] J. Wileden, "DREAM Design Notation Example: Scheduler for a Multiprocessor System," Dept. of Comp. and Comm. Science, Univ. of Michigan, Ann Arbor, RSSM/51, August 1977.

[37] J. Wileden, "DREAM – An Approach to Designing Large Scale, Concurrent Software Systems," *Proc. 1979 National Conference of the ACM*, 88–94, October 1979.

[38] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," *Journal of Digital Systems*, 177–197, Summer 1980.

[39] J. Wileden, J. Sayler, W. Riddle, A. Segal and A. Stavely, "Behavior Specification in a Software Design System," *Journal of Comp. Systems and Software*, vol. 3, no. 2, 123–135, June 1983.

[40] Y. Wong, "A DYMOL Simulation System," Dept. of Comp. and Info. Science, Univ. of Massachusetts, Amherst, SDLM/84-4, August 1984.

APPENDIX

In this appendix, we show how some of the analysis described in section 4 would be carried out. Our intention is to convey the flavor of the process, rather than to present a complete analysis of the system of section 4, and we therefore discuss only a few representative portions of the analysis in detail.

The constrained expression discussed in section 4 is derived directly from the design of Figure 1. For the analysis that follows, however, we use a constrained expression that is equivalent to the derived one, in the sense that it produces the same interpreted language. The new constrained expression is obtained in a purely mechanical fashion from the derived expression by applying the reduction algorithm described in [9]. The system expression for this new constrained expression is the interleave of the task expressions shown in Figures A-1 and A-2. The constraints required for this analysis are obtained by instantiating the constraint templates shown in Figure A-3. The resulting constrained expression has the property that every constrained prefix is a full string in the language of its system expression. Producing this constrained expression is the first step in the simplification process described in section 4.

We begin the analysis by identifying certain alternatives that can be deleted from the task expressions of Figures A-1 and A-2 because the event sequences represented by those alternatives cannot occur in system behaviors. At present, we do not know how to automate this part of the process, although certain aspects of it could obviously benefit from support by appropriate tools.

As an example, we show how the alternatives F_i-2 and F_i-3 in Figure A-1 can be eliminated. The alternative F_i-2 represents a sequence of events in which philosopher P_i picks up fork F_i and then philosopher P_{i-1} puts it down, while F_i-3 represents a sequence of events in which P_{i-1} picks up the fork and P_i puts it down.

We can write the task expression for F_i as

$$(F_i\text{-}1 \vee F_i\text{-}2 \vee F_i\text{-}3 \vee F_i\text{-}4)^*(F_i\text{-}5 \vee F_i\text{-}6 \vee F_i\text{-}7).$$

Suppose that a sequence of symbols associated with F_i-2 or F_i-3 occurs in a constrained prefix. Projecting such a constrained prefix onto the alphabet of the task expression F_i

Task expression for F_i:

$$
\begin{array}{ll}
\mathrm{F}_i\text{-1} & \Big(\textit{beg\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{beg\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{D})\,\textit{end\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{D}) \\
\mathrm{F}_i\text{-2} & \qquad \vee\ \textit{beg\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{beg\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{D})\,\textit{end\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{D}) \\
\mathrm{F}_i\text{-3} & \qquad \vee\ \textit{beg\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{beg\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{D})\,\textit{end\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{D}) \\
\mathrm{F}_i\text{-4} & \qquad \vee\ \textit{beg\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{beg\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{D})\,\textit{end\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{D})\Big)^* \\
\mathrm{F}_i\text{-5} & \quad \Big(\textit{starve}_a(\mathrm{F}_i.\mathrm{U}) \\
\mathrm{F}_i\text{-6} & \qquad\quad \vee\ \textit{beg\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_i;\mathrm{F}_i.\mathrm{U})\,\textit{starve}_a(\mathrm{F}_i.\mathrm{D}) \\
\mathrm{F}_i\text{-7} & \qquad\quad \vee\ \textit{beg\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{end\_rend}(\mathrm{P}_{i-1};\mathrm{F}_i.\mathrm{U})\,\textit{starve}_a(\mathrm{F}_i.\mathrm{D})\Big)
\end{array}
$$

Figure A-1
Task expression for F_i after reduction

Task expression for $\mathbf{P}_i$:

$\mathbf{P}_i$-1 $$\Big(\mathit{inv}(P_i;\mathrm{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{call}(P_i;F_i.U)\,\mathit{resume}(P_i;F_i.U)$$

$$\mathit{inv}(P_i;\mathrm{Eat})\,\mathit{call}(P_i;F_i.D)\,\mathit{resume}(P_i;F_i.D)\,\mathit{call}(P_i;F_{i+1}.D)\,\mathit{resume}(P_i;F_{i+1}.D)\Big)^*$$

$\mathbf{P}_i$-2 $$\Big(\mathit{inv}(P_i;\mathrm{Think})\,\mathit{starve}_c(P_i;F_{i+1}.U)$$

$\mathbf{P}_i$-3 $$\vee\ \mathit{inv}(P_i;\mathrm{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{starve}_c(P_i;F_i.U)$$

$\mathbf{P}_i$-4 $$\vee\ \mathit{inv}(P_i;\mathrm{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{call}(P_i;F_i.U)\,\mathit{resume}(P_i;F_i.U)$$

$$\mathit{inv}(P_i;\mathrm{Eat})\,\mathit{starve}_c(P_i;F_i.D)$$

$\mathbf{P}_i$-5 $$\vee\ \mathit{inv}(P_i;\mathrm{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{call}(P_i;F_i.U)\,\mathit{resume}(P_i;F_i.U)$$

$$\mathit{inv}(P_i;\mathrm{Eat})\,\mathit{call}(P_i;F_i.D)\,\mathit{resume}(P_i;F_i.D)\,\mathit{starve}_c(P_i;F_{i+1}.D)$$

$\mathbf{P}_i$-6 $$\vee\ \mathit{comp}(P_i)\Big)$$

Figure A-2
Task expression for $\mathbf{P}_i$ after reduction

For each task T:

$$\kappa_{nc}(T) = \lambda$$

For each entry $T_1.E$ and each task T_2 that calls $T_1.E$:

$$\kappa_{starve}(T_2; T_1.E) = starve_c(T_2; T_1.E) \vee starve_a(T_1.E) \vee \lambda$$

For each task T:

$$\kappa_{comp}(T) = \bigvee_{T'.E} starve_c(T; T_1.E) \vee \bigvee_{T.E} starve_a(T.E) \vee comp(T)$$

For each entry $T_1.E$ and each task T_2 that calls $T_1.E$:

$$\kappa_{synch}(T_2; T_1.E) =$$

$$\left(call(T_2; T_1.E)\, beg\_rend(T_2; T_1.E)\, end\_rend(T_2; T_1.E)\, resume(T_2; T_1.E) \right)^*$$

where $T'.E$ ranges over the entries called by T and $T.E$ ranges over all entries of T, the constraint alphabet for $\kappa_{nc}(T)$ is the set $\{ nc(T) \}$, and the constraint alphabets for $\kappa_{starve}(T_2; T_1.E)$, $\kappa_{comp}(T)$, and $\kappa_{synch}(T_2; T_1.E)$ consist of the symbols that appear in the corresponding constraints.

Figure A-3
Constraint templates required for the analysis

by erasing all the symbols not occurring in the task expression, we obtain a string from the language of

$$(F_i\text{-}1 \vee F_i\text{-}4)^*(F_i\text{-}2 \vee F_i\text{-}3)(F_i\text{-}1 \vee F_i\text{-}2 \vee F_i\text{-}3 \vee F_i\text{-}4)^*(F_i\text{-}5 \vee F_i\text{-}6 \vee F_i\text{-}7).$$

We give the argument for the case in which this string lies in the language of

$$(F_i\text{-}1 \vee F_i\text{-}4)^*\, F_i\text{-}2(F_i\text{-}1 \vee F_i\text{-}2 \vee F_i\text{-}3 \vee F_i\text{-}4)^*(F_i\text{-}5 \vee F_i\text{-}6 \vee F_i\text{-}7).$$

The argument for the other case is similar.

If any such constrained prefix exists, one exists in which the *end_rend* and *resume* symbols corresponding to the completion of the same rendesvous are adjacent. Hence, we may assume without loss of generality that the constrained prefix under consideration has this property. Consider the initial segment s of the constrained prefix ending with the adjacent $\mathit{end\_rend}(P_{i-1}; F_i.D)$ - $\mathit{resume}(P_{i-1}; F_i.D)$ pair coming from the first appearance of F_i-2.

Projecting s onto the union of the alphabets of the constraints $\kappa_{synch}(P_{i-1}; F_i.D)$ and $\kappa_{synch}(P_{i-1}; F_i.U)$ gives a string in the language of

$$\begin{gathered}\Big(\mathit{beg\_rend}(P_{i-1}; F_i.U)\,\mathit{end\_rend}(P_{i-1}; F_i.U)\,\mathit{beg\_rend}(P_{i-1}; F_i.D)\,\mathit{end\_rend}(P_{i-1}; F_i.D)\Big)^n \\ \mathit{beg\_rend}(P_{i-1}; F_i.D)\,\mathit{end\_rend}(P_{i-1}; F_i.D) \\ \triangle \\ \Big(\mathit{call}(P_{i-1}; F_i.U)\,\mathit{resume}(P_{i-1}; F_i.U)\,\mathit{call}(P_{i-1}; F_i.D)\,\mathit{resume}(P_{i-1}; F_i.D)\Big)^m\end{gathered}$$

for some $n, m \geq 0$. (For a regular expression e, we use the notation e^k to represent the concatenation of k copies of e.)

Note that the constraint $\kappa_{synch}(P_{i-1}; F_i.U)$ implies that there are the same number of $\mathit{end\_rend}(P_{i-1}; F_i.U)$ symbols as $\mathit{resume}(P_{i-1}; F_i.U)$ symbols in any initial segment of a constrained prefix ending in a $\mathit{resume}(P_{i-1}; F_i.U)$. Thus, we must have $n = m$. Similarly, $\kappa_{synch}(P_{i-1}; F_i.D)$ implies that $n + 1 = m$. This contradiction shows that no such constrained prefix exists. Hence alternatives F_i-2 and F_i-3 can be eliminated from the task expression.

Arguments of a similar nature can be used to eliminate alternative F_i-6 from the task expression for fork F_i and alternatives P_i-2, P_i-4, and P_i-5 from the task expression for philosopher P_i, producing the task expressions shown in Figure A-4. This analysis shows that, for example, a philospher holding both forks eventually puts them both down (since P_i-4 and P_i-5 are eliminated) and that a philosopher can eventually pick up the left fork (since P_i-2 is eliminated). Thus, the only way philosopher P_i can starve is while trying to pick up fork F_i.

As an example of the sort of additional analysis that would be carried out, we show that philosopher P_i starves only if philosopher P_{i-1} does. (This implies, of course, that one philosopher starves if and only if they all do.)

Suppose that there is a constrained prefix u containing a $starve_c(P_i; F_i.U)$ symbol. Using $\kappa_{starve}(F_i.U)$, we note that there is no $starve_a(F_i.U)$ symbol in the prefix u. Then alternative F_i-7 must contribute to u. Consider the projection of u on the union of the alphabets of the constraints $\kappa_{synch}(P_{i-1}; F_i.U)$ and $\kappa_{synch}(P_{i-1}; F_i.D)$. This projection must lie in the language of

$$\begin{gathered}\Big(call(P_{i-1}; F_i.U)\,resume(P_{i-1}; F_i.U)\,call(P_{i-1}; F_i.D)\,resume(P_{i-1}; F_i.D)\Big)^j \\ \Big(call(P_{i-1}; F_i.U)\,resume(P_{i-1}; F_i.U) \vee \lambda\Big) \\ \Delta \\ \Big((beg\_rend(P_{i-1}; F_i.U)\,end\_rend(P_{i-1}; F_i.U)\,beg\_rend(P_{i-1}; F_i.D)\,end\_rend(P_{i-1}; F_i.D)\Big)^k \\ beg\_rend(P_{i-1}; F_i.U)\,end\_rend(P_{i-1}; F_i.U)\end{gathered}$$

for some $j, k \geq 0$, where the first alternative in the line above the Δ corresponds to the case when P_{i-1}-3 contributes to u and the second corresponds to the case when P_{i-1}-6 contributes to u. The constraint $\kappa_{synch}(P_{i-1}; F_i.D)$ implies that $j = k$. The constraint $\kappa_{synch}(P_{i-1}; F_i.U)$ implies that the number of $call(P_{i-1}; F_i.U)$ symbols must be equal to the number of $beg\_rend(P_{i-1}; F_i.U)$ symbols. In the case where P_{i-1}-6 contributes to u, the former is j, while the latter is always $k+1$. Since $j = k$, this case is impossible. Hence P_{i-1}-3 contributes to u and u contains a $starve_a(P_{i-1}; F_{i-1}.U)$ symbol. Philosopher P_{i-1} thus starves attempting to pick up fork F_{i-1}.

Task expression for $\mathbf{F}_i$:

$\mathbf{F}_i$-1 $\Big(\mathit{beg\_send}(P_i;F_i.U)\,\mathit{end\_send}(P_i;F_i.U)\,\mathit{beg\_send}(P_i;F_i.D)\,\mathit{end\_send}(P_i;F_i.D)$

$\mathbf{F}_i$-4 $\vee\ \mathit{beg\_send}(P_{i-1};F_i.U)\,\mathit{end\_send}(P_{i-1};F_i.U)\,\mathit{beg\_send}(P_{i-1};F_i.D)\,\mathit{end\_send}(P_{i-1};F_i.D)\Big)^*$

$\mathbf{F}_i$-5 $\Big(\mathit{starve}_a(F_i.U)$

$\mathbf{F}_i$-7 $\vee\ \mathit{beg\_send}(P_{i-1};F_i.U)\,\mathit{end\_send}(P_{i-1};F_i.U)\,\mathit{starve}_a(F_i.D)\Big)$

Task expression for $\mathbf{P}_i$:

$\mathbf{P}_i$-1 $\Big(\mathit{inv}(P_i;\text{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{call}(P_i;F_i.U)\,\mathit{resume}(P_i;F_i.U)$

$\mathit{inv}(P_i;\text{Eat})\,\mathit{call}(P_i;F_i.D)\,\mathit{resume}(P_i;F_i.D)\,\mathit{call}(P_i;F_{i+1}.D)\,\mathit{resume}(P_i;F_{i+1}.D)\Big)^*$

$\mathbf{P}_i$-3 $\Big(\mathit{inv}(P_i;\text{Think})\,\mathit{call}(P_i;F_{i+1}.U)\,\mathit{resume}(P_i;F_{i+1}.U)\,\mathit{starve}_c(P_i;F_i.U)$

$\mathbf{P}_i$-6 $\vee\ \mathit{comp}(P_i)\Big)$

Figure A-4
Task expressions after further simplification

Similar arguments show that the forks are used by the philosophers in a mutually exclusive fashion.

The analysis described here depends, of course, on features of the particular constrained expression being analyzed, and therefore on features of the system represented by that constrained expression and the semantics of the notation used to design that system. In this example, we made heavy use of the constraints which ensure proper synchronization of rendezvous. If the dining philosophers system had been designed using a notation employing buffered message transmission, the corresponding constraints would ensure that the number of messages received from a communication channel is always less than or equal to the number of messages sent. In this case, the corresponding analysis would have involved inequalities rather than equations.